CORRECTED VERSION*

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : H04L 29/00

A2

(11) International Publication Number: WO 00/19681

(43) International Publication Date: 6 April 2000 (06.04.00)

(21) International Application Number: PCT/US99/22523

(22) International Filing Date: 30 September 1999 (30.09.99)

(30) Priority Data:
09/163,906 30 September 1998 (30.09.98) US

(71) Applicant: XBIND, INC. [US/US]; Suite 13C, 55 Broad Street, New York, NY 10004 (US).

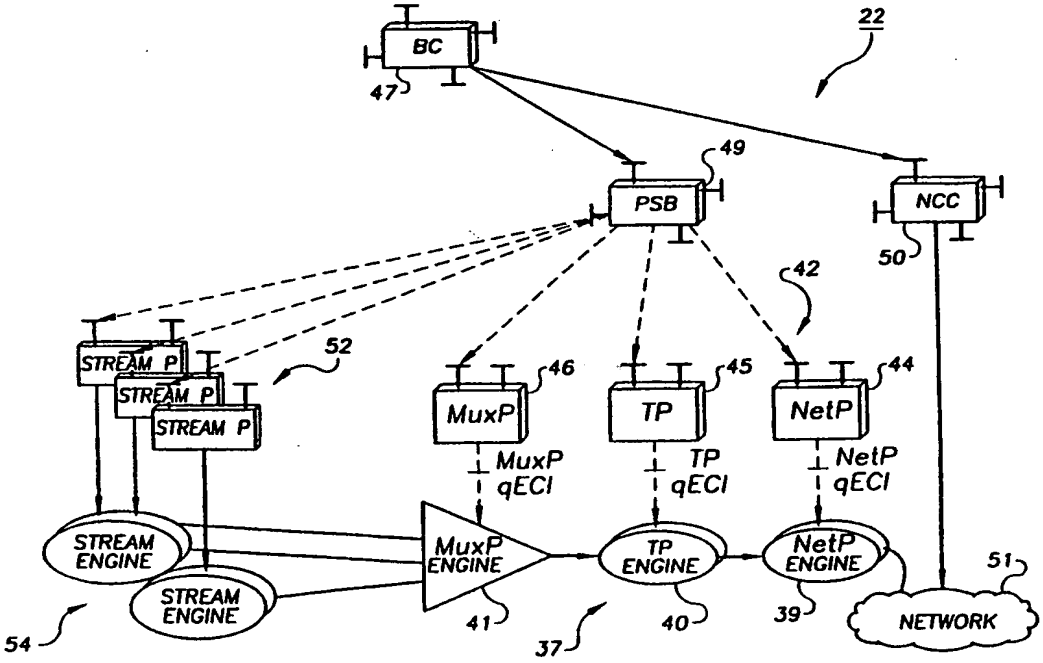
(72) Inventors: HUARD, Jean–François; Apartment #37, 200 Claremont Avenue, New York, NY 10027 (US). LAZAR, Aurel, A.; Suite 32A, 410 Riverside Drive, New York, NY 10025 (US).

(74) Agent: NUGENT, Elizabeth, E.; Choate, Hall & Stewart, Exchange Place, 53 State Street, Boston, MA 02109 (US).

(81) Designated States: AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: SYSTEM FOR BUILDING AND DYNAMICALLY RECONFIGURING PROTOCOL STACKS



(57) Abstract

A system which builds customized network protocol stacks uses plural computer–executable engines located in a data path, at least one computer–executable controller, and plural computer–executable front–ends interposed between the engines and the at least one controller. The system binds the at least one controller to selected front–ends, and instructs the selected front–ends to bind to corresponding engines in the data path so as to create a communication path between the at least one controller and each corresponding engine. The corresponding engines are then ordered to bind together so as to create a protocol stack within the data path.

*(Referred to in PCT Gazette No. 36/2000, Section II)

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

SYSTEM FOR BUILDING AND DYNAMICALLY RECONFIGURING PROTOCOL STACKS

BACKGROUND OF THE INVENTION

Field Of The Invention

The present invention is directed to a system which builds customized protocol stacks and which has the ability to reconfigure those protocol stacks dynamically so as, e.g., to upgrade a network connection or, more generally, to improve network quality of service.

Description Of The Related Art

In recent years, the centralized computing model, represented best by large mainframes computers, has given way to network systems with distributed processing capabilities. Such systems comprise plural devices, such as personal computers, network peripherals, and the like, linked together via network media. Communications exchanged between these devices are typically in the form of data packets, and are effected according to one or more predefined network protocols.

To implement communications between such devices, each device contains a series of hardware and software layers designed to perform specific functions which enable data to pass between the devices. This series of hardware and software layers is known as a protocol stack. To transmit data from one device to another, a transmitting application sends the data "down" the protocol stack. The data passes through each of the stack's layers, and is acted upon by each layer which, e.g., adds information needed to forward the data to its destination. At the receiving device, the data traverses a similar protocol stack, this time in reverse, until the data reaches its final destination.

One protocol stack model, promulgated by the International Standards Organization ("ISO"), is comprised of seven layers. As shown in Figure 1, these seven layers include the physical, data link, network,

transport, session, presentation and application layers. The various layers operate on data received from, or going to, the physical layer (i.e., the network medium). In brief, the data link layer effectively removes errors from data transmitted over the network medium; the network layer routes data packets by selecting a path over the network medium; the transport layer divides a data stream from the session layer and passes the data to the network layer; the session layer sets up connections between two network devices and controls transfer of data therebetween; the presentation layer operates on the data so that it can be presented to a user; and the application layer contains protocols which allow the data to be transmitted to an application running on a network device.

Another well-known protocol stack model is known as Transmission Control Protocol/Internet Protocol ("TCP/IP"). As shown in Figure 2, this protocol stack includes only four layers: the link layer, the network layer, the transport layer, and the application layer. In brief, the link layer is responsible for communicating with actual network hardware (e.g., an Ethernet card). Data received by the link layer from the network layer is output to a network medium, while data received by the link layer from the network medium is provided to the network layer. The network layer determines how to get the data to its destination; the transport layer provides data flows for the application layer according to a predefined protocol; and the application layer allows users to interact with the network.

To effect communications over a particular network, a device must include a protocol stack which is right for that network. For example, to communicate over the Internet, a networked PC must include the TCP/IP protocol stack described above. To communicate over another type of network, such as a local area network ("LAN"), a protocol stack having a different structure (i.e., different types and/or numbers of layers), such as the OSI protocol stack described above, may be required. Conventionally, it is necessary to load a device with a different protocol stack for each network over which the device is to communicate. This is inefficient, particularly where memory space on the device is limited.

In addition to variations in structure (i.e., different types and/or numbers of layers) the same layer of a protocol stack may support different protocols, depending upon network and application considerations. For example, in the TCP/IP protocol stack described above, the transport layer may support the TCP or UDP protocol. Likewise, the network layer in that case may support either the IP or the ARP protocols. Conventionally, these protocols are "hard-coded" within the layers of the protocol stack. This can be disadvantageous for a number of reasons.

For example, the emergence of distributed multimedia applications exhibiting significantly more stringent Quality of Service ("QOS") requirements than conventional data-oriented applications oftentimes calls for new transport protocols with different characteristics to co-exist and be integrated within single applications. The different delivery requirements posed by these diverse multimedia applications often imply the need for highly customized protocol implementations. For at least these reasons, conventional predefined protocol stacks, such as those described above, may be insufficient to meet all of the needs of such distributed multimedia applications.

Accordingly, there exists a need for a way to build a customized protocol stack, both in terms of layers and in terms of protocols supported within those layers, which can be used to meet the communications requirements of various types of multimedia applications, and of various types of applications in general.

SUMMARY OF THE INVENTION

The present invention addresses the foregoing needs by providing an object-oriented transport architecture in which a variety of protocol layers (e.g., "engines") can be bound together dynamically, on a per call basis. By binding these protocol layers together, a protocol stack that meets the special needs of an application can be built. As a result, the present invention makes it possible to provide enhanced QOS, without storing a multitude of "hard-coded" protocol stacks. Moreover, the

invention reduces the need to prepare specialized protocol stacks to meet the varying requirements of different applications.

According to one aspect, the present invention is a system (i.e., a method, an apparatus, and computer-executable process steps) for building customized network protocol stacks using plural engines located in a data path, at least one controller, and plural front-ends interposed between the engines and the at least one controller. In operation, the controller binds to selected front-ends, and instructs the selected front-ends to bind to corresponding engines in the data path so as to create a communication path between the controller and each corresponding engine. Thereafter, via each communication path, the controller orders the corresponding engines to bind together so as to create a protocol stack within the data path.

In preferred embodiments of the invention, the engines comprise plural types of network provisioning engines, plural types of transport protocol engines, and multiplexing engines. In these embodiments, the front ends are selected based on whether a network provisioning engine, a transport protocol engine, and/or a multiplexing engine is to be bound to the controller. By including different types of each engine, it is possible to build a protocol stack which has both differing numbers and types of layers. As a result, the invention provides increased flexibility in building customized protocol stacks.

In other preferred embodiments of the invention, the protocol stack builder selects which, if any, of the network provisioning engines, the transport protocol engines, and the multiplexing engines are to be bound to the selected front-ends based on either a user-input command, a remote network command, or a desired QOS for a network connection. Thus, using the present invention, it is possible to build and/or reconfigure protocol stacks either locally, remotely or automatically. For example, in a case that particular QOS parameters have been input to the invention, the invention will monitor the actual QOS parameters for a network connection. In the event that the actual QOS parameters deviate significantly from the input QOS parameters, the invention will

automatically reconfigure the protocol stack so that the actual QOS parameters fall within an acceptable range of the input QOS parameters. In other embodiments, on the other hand, the invention may reconfigure the protocol stack in response to a local or remote command, irrespective of the network QOS.

In particularly preferred embodiments, the protocol stack builder has the ability to bind the controller to one or more media processor front-ends, and to instruct the media processor front-ends to bind to corresponding media processor engines. This creates a communication path between the controller and each corresponding engine. The controller then may order, via each communication path, the corresponding media processor engines to bind together within the data path. By virtue of this feature, it possible to cascade various media processing applications in a simple and efficient manner.

According to another aspect, the present invention is a system for binding a media transporter engine in a data path to a media processor engine, which uses at least one computer-executable controller, and at least one computer-executable front-end interposed between the media transporter engine and the at least one controller. The system binds at least one controller to a selected front-end, and instructs the selected front-end to bind to a corresponding media transporter engine in the data path so as to create a communication path between the at least one controller and the corresponding media transporter engine. The system then orders, via the communication path, the corresponding media transporter engine in the data path to bind to the media processor engine.

This brief summary has been provided so that the nature of the invention may be understood quickly. A more complete understanding of the invention can be obtained by reference to the following detailed description of the preferred embodiment thereof in connection with the attached drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 shows the ISO standard protocol stack.

Figure 2 shows the standard TCP/IP protocol stack.

Figure 3 shows a computer system on which the present invention may be implemented.

Figure 4 shows the architecture of the computer system shown in Figure 3.

Figure 5 shows components of the programmable transport architecture of the present invention.

Figure 6 shows an abstraction of an engine used in the invention.

Figure 7 depicts, graphically, the requirements for binding two engines together.

Figure 8 shows an abstract of a front-end used in the invention.

Figure 9 depicts, graphically, the interaction between the engines and the front-ends in accordance with the invention.

Figure 10 shows an abstract of a binding controller used in the invention.

Figure 11 shows the architecture of an MPEG-4 DMIF which incorporates the protocol stack builder of the present invention.

Figure 12 shows a graphical user interface for a video conferencing application used with the present invention.

Figure 13 shows a graphical user interface into which a user may input QOS requirements for the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

The present invention comprises computer-executable code (i.e., process steps) which builds and which dynamically reconfigures network protocol stacks. The invention can be used in any type of electronic device, including, but not limited to, desk-top personal computers, laptops, digital telephones, digital televisions, and video-conferencing equipment. For the sake of brevity, however, the invention will be described in the context of a networked desk-top computer system

only.

Figure 3 shows a representative embodiment of a computer system 1 on which the invention may be implemented. As shown in Figure 3, personal computer ("PC") 2 includes network connection 4 for interfacing to a network, such as the Internet, an ATM network, or the like, and fax/modem connection 6 for interfacing with other remote devices such as a digital camera, digital video camera, and the like. PC 2 also includes display screen 7 for displaying information to a user, keyboard 9 for inputting text and user commands, mouse 10 for positioning a cursor on display screen 7 and for inputting user commands, disk drive 11 for reading from and writing to floppy disks installed therein, and DVD ("digital video disk") drive 12 for accessing information stored on DVD. PC 2 may also have one or more peripheral devices (not shown) attached thereto for inputting text, graphics images, or the like, and printer 14 attached thereto for outputting images.

Figure 4 shows the internal structure of PC 2. As shown in Figure 4, PC 2 includes memory 16, which comprises a computer-readable medium such as a computer hard disk. Memory 16 stores data 17, applications 19, print driver 20 and operating system 21. In preferred embodiments of the invention, operating system 21 is a windowing operating system, such as Microsoft® Windows98 or Microsoft® Windows NT versions 4.0 or 5.0; although the invention may be used with other operating systems as well. Among the applications stored in memory 16 is computer-code to implement components of the present invention, namely programmable transport architecture 22. In brief, programmable transport architecture 22 is comprised of computer-executable process steps which build and which dynamically reconfigure a protocol stack for PC 2 based on one or more predetermined parameters, such as QOS or the like. In the preferred embodiment of the present invention, programmable transport architecture 22 includes of a number of modules referred to below as "engines", "front-ends", and "controllers", which are also stored in memory 16. A detailed description of programmable transport architecture 22 and these modules is provided below.

Also included in PC 2 are display interface 24, keyboard interface 26, mouse interface 27, disk drive interface 29, DVD drive interface 30, computer bus 31, RAM 32, processor 34, and printer interface 36. Processor 34 preferably comprises a microprocessor or the like for executing applications, such those noted above, out of RAM 32. Such applications, including programmable transport architecture 22, may be stored in memory 16 (as noted above) or, alternatively, on a floppy disk in disk drive 11 or a DVD in DVD drive 12. Processor 34 accesses applications (or other data) stored on a floppy disk via disk drive interface 29, and accesses applications (or other data) stored on a DVD via DVD drive interface 30.

Application execution and other tasks of PC 2 may be initiated using keyboard 9 or mouse 10, commands from which are transmitted to processor 34 via keyboard interface 26 and mouse interface 27, respectively. Similarly, application execution may be initiated remotely via, e.g., network interface 4 and a network interface card (not shown). Output results from applications running on PC 2 may be processed by display interface 24 and then output via network connection 4 or, alternatively, displayed to a user on display 7. To this end, display interface 24 preferably comprises a display processor for forming video images based on video data provided by processor 34 over computer bus 31, and for outputting those images to display 7. Output results from other applications, such as word processing programs, running on PC 2 may be provided to printer 14 via printer interface 36. Processor 34 executes print driver 20 so as to perform appropriate formatting of such print jobs prior to their transmission to printer 14.

Turning to programmable transport architecture 22, this application has an object-oriented transport architecture in which the atomic processing entity is based on the consumer/producer paradigm. Specifically, programmable transport architecture 22 includes computer-executable media transporter components (described below) that are separately represented by their transport abstractions, called engines, their control and management abstractions, called front ends, and a set of

controllers implementing network services, such as dynamic building of protocol stacks.

In brief, the engines are located in the data path of a multimedia communication session and are responsible for data processing (including protocol implementation), multiplexing, and scheduling of channels. The engines also interact with the operating system when transferring media flows through network interface cards. The front-ends control and manage engine resources to effect resource provisioning, accounting and QOS control, among other things. The controllers provide network services to an application programmer and thereby off-load work from any multimedia applications. By encapsulating domain specific knowledge, the controllers reduce the amount of technical knowledge required by an application developer to create multimedia applications. A more detailed description of the engines, the front-ends, and the controllers is provided below.

1.0 Architecture

As noted above, the architecture of the preferred embodiment of the present invention is based on the consumer/producer model. In general, consumer/producer components can be classified into two categories, namely media processors and media transporters. Media processors refer to components that are capable of transforming a media stream (i.e., a data stream for a multimedia presentation) from one format into another format by processing the contents of the stream. Examples of well-known media processors include transcoders and encryption devices.

A preferred architecture for the present invention includes two types of media processors: a streamed device and an encryptor. Streamed devices include media stream producers and media stream consumers. Media stream producers "grab" data using a physical device, such as a camera or a microphone, and then compress the data to generate encoded data. An example of such a media stream producer is a PC video card that compresses raw video data into MPEG-4 (or, alternatively MPEG-2, MPEG-1, DVB, etc.) encoded video data. Media stream

consumers decode encoded video data and render the decoded video data to a physical device, such as display screen or a speaker. An example of a media stream consumer is a digital signal processor chip set that processes, and plays back, encoded audio. Finally, encryptors either encrypt or decrypt data depending on whether they are located on the sender or receiver side of the communication channel.

Media transporters comprise components of a protocol stack which carry and route media streams without discerning or altering the character of the multimedia data in those streams. In this regard, media transporters implement transport functionalities such as scheduling of channels, multiplexing, flow control, encapsulation and de-encapsulation, segmentation and re-assembly, etc. On the transmitting side of a channel, media transporters add a header to each data unit of a message being sent to the receiving side of the channel. Media transporters on the transmitting side may also fragment a message if that message is too large to be transmitted as is. On the receiving side of the channel, media transporters reassemble messages, and de-encapsulate and forward data units therein to a next component in a protocol stack on the receiving side. In general, a chain of media transporters is traversed until the data reaches a media processor, such as the streamed devices described above, that renders the data back or saves the data in a file for future use. The present invention, among other things, links appropriate media transporters to form a protocol stack based upon one or more events or factors, such as a degradation in QOS, detection of a user-input command, etc., or based upon a pre-set default stack configuration for a particular network connection.

1.1 Architecture Overview

Figure 5 depicts a preferred architecture for the present invention. Three layers of components are shown in Figure 5. First, media transporter engines 37 are located in the data path at the bottom layer. These engines include network provisioning ("NetP") engine 39 (which roughly corresponds to the "network" layer of the generalized OSI protocol stack), transport protocol ("TP") engine 40 (which roughly

corresponds to the "transport" layer of the generalized OSI protocol stack), and multiplexing ("MuxP") engine 41 (which roughly corresponds to the "session" layer of the generalized OSI protocol stack). Descriptions of these engines are provided below. Front-ends 42, which allow for media control and management, are located in the middle layer. These front-ends include NetP front-end 44, TP front-end 45, and MuxP front-end 46. Binding controller 47 is located on the top layer, and interacts with protocol stack builder controller ("PSB") 49 and network connection controller ("NCC") 50. As described in more detail below, PSB 49 acts directly on the front-ends so as to manage and control the generation, consumption, and processing of media streams, and NCC 50 establishes and maintains network connections between end-systems over network 51.

As noted, the architecture shown in Figure 3 allows for dynamic creation of protocol stacks by binding engines at run-time. The architecture also facilitates complex multimedia stream processing operations by using media processor front-ends (i.e., "StreamPs" 52) to bind together one or more media processor engines (i.e., stream engines 54), which process data and output the processed data to the protocol stack. Binding of media processor engines is similar to the process for binding engines described below. Examples of media processor engines which may be used with the present invention include, but are not limited to, H.261, H.263, H.263+, MPV, motion JPEG video, MPEG-1 system stream, MPEG-2 transport and program stream, and (for audio) 64 kbps encoded PCM, u-law and A-law.

In addition to the foregoing, the architecture allows for multiple implementations of a media transporter engine to co-exist and for new capabilities (e.g., new consumer/producers) to be added without having to modify components of the architecture significantly. This, of course, differs from traditional transport architectures which utilize pre-installed transport protocol stacks that cannot be customized.

1.2 Consumer/Producer Engines

Consumer/producer engines ("CPGs") are computer-

executable components that implement protocol state machines and that have the ability to support and to schedule multiple channels at the same time. CPGs are implemented in user space (as opposed to operating system kernel space) and are generally hardware independent, but may be dependent on the operating system's multi-thread programming support and networking support. In the present invention, CPGs are located in the communication data path, as shown in Figure 5. One example of a CPG is "qStack", which is a lightweight transport protocol suitable for real-time, interactive communications. See M. C. Chan, J.-F. Huard, A. A. Lazar, and K.S. Lim, "On Realizing a Broadband Kernel for Multimedia Networks", in Proc. of the Third COST 237 Workshop on Multimedia Telecommunications and Applications, Barcelona, Spain (November 25-27, 1996), the contents of which are hereby incorporated by reference into the subject application as if set forth herein in full.

Figure 6 is an abstraction of a CPG used in the present invention. As shown in Figure 6, CPG 56 has two interfaces: a QOS-based control interface ("qECI") and a media transfer interface ("MTI"). The qECI is visible only to the front-ends that are bound to it and that are using its specific services. In this regard, by convention, the qECI of an engine is named using the front-end acronym of the engine followed by qECI. For example, the transport protocol engine ("TP") control interface is denoted by TP qECI.

The engine's MTI is used to move data from one CPG to another CPG which is interfaced thereto. Additionally, the MTI provides some I/O control capabilities. Specifically, the MTI can be used to query for an engine's maximum service and protocol data unit sizes, to set a blocking mode, to flush the engine's buffer, etc. The MTI is generally visible only to adjacent engines in the data path. Only one MTI is shown in Figure 6, since the same MTI is used both to send and to receive data from an adjacent engine. To this end, an opaque handle (e.g, a channel identifier) is provided to distinguish between each channel.

As opposed to the qECI, which is specialized for each

engine, the MTI is common to all engines. As a result, any engine can be connected to any other engine. A variety of different protocol stacks can therefore be built simply by connecting the appropriate engine via their MTIs. When a data unit is transferred via the MTI, e.g., via a send or a receive command, the engine processes the data unit and schedules its transfer to an adjacent engine in the data path. In a case that the engine of interest is a NetP engine, the engine schedules the data units to be sent out into the network. When connecting (i.e., binding) engines, appropriate scheduling and elimination of unnecessary data copies should be considered in order to achieve satisfactory overall performance and throughput.

Figure 7 shows the details of binding together two engines A and B. In this regard, the process of binding engines comprises the exchange, between the two engines, of an indirect pointer (i.e., a pointer to a pointer) to an opaque handle (i.e., a channel identifier) and an indirect pointer to a procedure table in the subsequent engine. Thus, to bind engine B to engine A, engine B is provided with an indirect pointer to engine A's procedure table and an indirect pointer to the opaque handle for the desired channel. Once the opaque handle and the procedure table are accessed by engine B, engine B is able to invoke procedures, or functions, via the table of engine A for that particular channel. These procedures include, among other things, capabilities to send data to, and receive data from, engine A. The Appendix attached hereto shows representative examples, written in the ANSI C language, of a pointer to an entry point of procedure table, a procedure table, and an indirect pointer to the procedure table. Of course, those of ordinary skill in the art will understand that there are a number of variations to the examples shown in the Appendix.

1.3 Consumer/Producer Front-Ends

In general, consumer/producer front-ends ("CPFs") define an abstraction layer which permits a signaling system to control the heterogeneous transport and computing resources of an end-system. CPFs have open interfaces such that each CPF abstracts a single communication segment (e.g., data stream) of an end-to-end network channel.

In this regard, in the context of the present invention, CPFs are used by controllers (i.e., the signaling system) to control and to manage CPGs 37 (i.e., the end engines). More specifically, CPFs provide a communication path between the engines and the controller, which communication path also includes logic that performs any data format conversions necessary to permit the engines and the controller to communicate with each other. Thus, as shown in Figure 5, a controller (i.e., PSB 49) controls TP engine 40 via TP front-end 45, the controller controls NetP engine 39 via NetP front-end 44, and the controller controls MuxP engine 41 via MuxP front-end 46. In addition, as described in more detail in section 1.4 below, multiple front-ends can be attached to a single engine, since each engine has the capability to support multiple channels. As a result, in some embodiments of the invention, the controller may control different channels in a single engine via different front-ends. At this point, it is noted that PSB 49 may be local to the front-ends (e.g., on the same machine), or remote therefrom (e.g., at a different geographic location on a network).

Figure 8 is an abstraction of a CPF used in the present invention. As shown in Figure 8, CPF 57 has three interfaces. These interfaces preferably comprise QOS-based application programming interfaces ("APIs") used by the binding controllers and management system to control and manage the engines. The control and management interfaces, labeled "C" and "M", respectively, provide controllers with the ability to control and manage channels abstracted by the front ends. For example, via the C interface, a controller can establish and release a channel to an engine in order to effect resource provisioning, accounting, and QOS control (e.g., QOS monitoring, QOS violation detection, QOS adaptation, and QOS re-negotiation). Via the M interface, controllers can dynamically load and configure an engine. The front-end's transport interface, labeled "T", is used for binding of the engines (i.e., obtaining the engine's MTI, and obtaining a handle of an associated communication channel) and for negotiating media transfer parameters, such as a maximum protocol data unit size.

CPFs thus provide a way of interfacing a controller to an engine that implements the transport functionality of the consumer/producer. To this end, the front-ends use dynamically linked libraries that allow them to bind to the engines at run-time. To bind to an engine, a front-end need only know the type of the engine (e.g., TP, NetP, etc.). Which of a particular type of engine the front-end binds to is determined by PSB 49. For example, front-end 45 may bind to a TCP TP engine, an XTP TP engine, an RTP TP engine, etc.; whereas front-end 44 may bind to a UDP/IP NetP engine, an AAL5/ATM NetP engine, etc. As alluded to above, the controller and an engine do not need to use the same language in order to communicate, as they each have access to a respective middleware front-end interface. In order to provide a controller with transparent access to an engine via the front-end, the engine need only support the functions defined by its qECI. In this regard, most control operations (effected via the C interface) are mapped to a qECI operation, while the M interface is used mainly to manage the front-end.

Interfaces from a collection of consumer/producer front-ends form a repository called the Binding Interface Base (BIB), from which a controller may select a front-end. See Lazar, A. A., "Programming Telecommunication Networks", IEEE Networks, pgs. 8-18 (September/October 1997), the contents of which are hereby incorporated by reference into the subject application as if set forth herein in full. Thus, in the preferred embodiment of the invention, the front-end interfaces shown in Figure 8 form part of the BIB for the invention.

1.4 Interaction Between Engines and Front-ends

The purpose of distinguishing between CPGs and CPFs above is to separate control over communications into two different time scales: the transport time scale (i.e., at the packet level) and the time scale of flows or virtual circuits (i.e., at the call level). This separation also permits remote control over individual transport connections. In this regard, as noted above, each CPG is capable of performing software multiplexing, and thus, may support multiple channels that are each

abstracted by a single CPF. In such a case, each CPF contains the specific state of the channel, e.g., the measured QOS of the channel. When channel segments are initialized, the CPG assigns an opaque handle to access the channel's state, and then provides that handle to a respective CPF. Thus, all CPFs that are bound to a CPG can access the capabilities of the CPG via the CPG's qECI and their respective opaque handle.

Figure 9 shows examples of interactions between CPGs and CPFs. In this case, CPG 60, which is the CPG under consideration, is bound to four other engines (i.e., CPGs A, B, C and D). As shown in Figure 9, CPG 60 supports three channels that are each abstracted by a different CPF (i.e., CPFs 61, 62 and 63). In the example shown in Figure 9, CPG-A and CPG-B are audio and video stream engines, CPG-C and CPG-D are AAL5/ATM and UDP/IP NetP engines, and CPG 60 comprises the qStack transport protocol engine noted above. As noted, qStack provides flow control and QOS support for two audio channel segments, one going though an IP network and one going through an ATM network, and for a video channel segment that goes through an ATM network.

In Figure 9, the three channel segments going through CPG 60 are labeled Id1, Id2 and Id3. The three front-ends bound to CPG 60 (namely, CPFs 61, 62 and 63) each abstracts and controls one of these channel segments, and maintains an opaque handle locally for its respective segment. Starting with segment Id1, an audio device connected to the ATM network has its qStack segment labeled with identifier Id1. As shown in the figure, the data path for segment Id1 comprises CPG-A to CPG 60 to CPG-C. Likewise, that segment is controlled by the CPF having identifier Id1 (in this case, CPF 61). CPG-A also maintains identifier Id1, which is used to provide data to qStack through the MTI of CPG 60. In accordance with the present invention, identifier Id1 was provided to CPG-A when the data path was created by programmable transport architecture 22. The other two segments, namely Id2 and Id3, are controlled in a similar manner. That is, CPF 62 abstracts segment Id2, which comprises an audio channel for an IP network. The data path for that segment comprises CPG-A to CPG 60 to CPG-D. Lastly, CPF 63

abstracts segment Id3, which comprises video communication for an ATM network. The data path for that segment comprises CPG-B to CPG 60 to CPG-C.

Thus, as described above, CPG 60 performs protocol processing for, and is responsible for multiplexing channels between, adjacent engines. Media transfer between respective engines is achieved via the MTI using the opaque handle that identifies the proper segment. At this point, it is noted that a detailed illustration of all protocol stack components would have included six additional front-ends in Figure 9: two attached to CPG-A to control the two audio streams, one attached to CPG-B to control the video stream, two attached to CPG-C to control the ATM connections, and one attached to CPG-D to control the UDP/IP channel.

1.5 Binding Controllers

Binding controllers ("BCs") control creation, operation, management, and programming of services. Their primary purpose is to off-load work from multimedia applications. That is, as opposed to front-ends, which represent low-level middleware services, BCs comprise high-level middleware controllers that an application developer would normally use to build a multimedia application. To this end, BCs encapsulate domain specific knowledge and thereby reduce the amount of technical knowledge required by application developers when writing multimedia applications. Functions performed by controllers include network connection control (NCC 50) and protocol stack building (PSB 49). Additional services, such as transport monitoring and QOS mapping may be available to the controllers via, e.g., BC 47 or an additional server. Collectively, plural binding controllers provide Broadband Kernel Services. See Lazar, A.A., "Programming Telecommunication Networks", cited above.

A BC is comprised of an algorithmic component and a data component. See Chan, et al., "On Realizing a Broadband Kernel for Multimedia Networks", cited above. The algorithmic component comprises execution logic for the service instance, while the data portion

comprises an abstraction of its state. As shown in the abstraction provided in Figure 10, BCs have four interfaces from which to control creation, operation, management, and programming thereof. These interfaces are open and allow all binding controllers to be invoked remotely. Briefly, the invocation interface is the entry point for the execution or instantiation of a service. The operation interface defines the operational functionality of the controller and allows for monitoring and manipulation of service instance states during execution. The operation interface is typically the primary interface of the controller. The programming interface permits logic in the controller to be manipulated and/or changed during execution of a service. Finally, the management interface permits monitoring of the controller's states and manipulation (e.g., modification) of parameters in the controller. The use of each interface in the context of the present invention is as follows.

More specifically, the invocation interface is the entry point for creating a useful protocol stack based on input QOS requirements and a given network endpoint (e.g., a VPI/VCI pair in an ATM connection). The operation interface permits changing the QOS associated with a protocol stack or even modifying the protocol stack while it is active. In this regard, a new QOS may be input by a user, e.g., via keyboard 9, or downloaded from a remote source, such as another network device at a remote network node. The programming interface allows the management system to configure the logic of the protocol stack builder; that is, the rules concerning how to create useful protocol stacks. Finally, the management interface permits the management system to manage the protocol stack builder, for example, by specifying the maximum number of channels that may be allowed.

2.0 Media Transporter Components

The present invention defines media processor and media transporter components of a protocol stack by setting up channel segments between interfaces of appropriate CPGs. The following describes defining a protocol stack using three media transporter components, namely, NetP,

TP and MuxP, using three transport services. Of course, the invention is not limited to using only these three media transporter components.

2.1 Network Provisioning

The network provisioning ("NetP") component, comprised of NetP engine 39 and NetP front-end 44, is responsible for transmitting data across networks. Specifically, NetP provides transparent transfer of data for the transport protocol component described below. In a protocol stack, NetP comprises the lowest level (i.e., the base layer) of the protocol stack and includes the user-plane functionality of traditional transport protocols.

NetP's control capabilities also include provisioning calls (e.g., opening/closing virtual circuits) and network QOS control. Its provisioning capability revolves around the creation and destruction of opaque network handles (e.g., sockets) associated with network endpoints. NetP's QOS control capability comprises pacing injection of packets into the network and monitoring packet QOS. NetP also has a management capability, which comprises setting network QOS requirements (when applicable) and accounting, i.e., counting a number of transferred packets.

NetP's front-end does not directly perform NetP's control capabilities, but rather maps them to its engine, marshaling arguments of the calls whenever needed. This control capability is typically realized by invoking system or library calls that communicate with a network interface card driver or the like. For the pacing capability described above, software interval timers might be used if not provided by the hardware. However, pacing is preferably provided by hardware, such as a network interface card, since software pacing is relatively inefficient. For the implementations of ATM and IP NetP, the MTIs simply invoke the equivalent system or library calls after performing some additional protocol processing.

In this regard, a variety of different NetP engines can be used with the present invention including, but not limited to, UDP/IP, AAL5/ATM and AAL1/ATM engines. Thus, for example, if a particular NetP engine is not providing a desired QOS, the invention can switch to a

different NetP engine which better meets the QOS requirements, with or without switching other engines (e.g., the TP engines). To effect such switching, PSB 49 need merely instruct the appropriate front-end to bind to a different NetP engine, thereby causing the first engine to be substituted, in the data path, with that other engine.

In preferred embodiments of the invention, a transport monitor service (described below) in BC controller 47 has the ability to monitor the QOS of the current network connection and to send appropriate event information in the case of a QOS violation. When a predetermined event occurs, such as a degraded QOS, a "remote" command received from a remote network device, or a user-input command, PSB 49 is thus able to construct a new protocol stack dynamically. PSB 49 does this by selecting another, more appropriate, NetP engine, and by substituting the current NetP engine with the other NetP engine, with or without altering any other aspects of the protocol stack. In much the same way, at run-time, a customized protocol stack can be constructed which provides for a desired QOS or, alternatively, based on a pre-set default for the network connection. As described below, the same can be done for transport protocol and multiplexer engines so as to provide for further variations in the protocol stack.

2.2 Transport Protocol

The transport protocol ("TP") component, comprised of TP engine 40 and TP front-end 45, is used to provide an end-to-end communication capability that has QOS support. TP is the first layer in the protocol stack that is end-to-end QOS-aware. If needed, the TP component has the ability to ensure reliable end-to-end communication. It relies on NetP to receive error free data, but may request retransmission if a segment of data is missing. Flow control resides in this layer to manage data flow through an established channel. This means that the TP media transporter assumes that a feedback channel is available for QOS adaptation and flow control.

TP provides end-to-end QOS control, management, and

accounting capabilities. More specifically, an end-to-end QOS control API is provided for setting TP parameters and for monitoring the QOS. The specific capability of the API depends on the TP specifications; however, the API is preferably common for all TPs. The management interface of the TP front-end permits selection of a TP engine associated with the channel. It can also be used to obtain accounting information such as call duration and an amount of recovered data delivered to the application. The TP front-end also performs slow time-scale monitoring of channels and initiates QOS re-negotiation procedures upon detection of sustained QOS violations. Finally, as described in more detail below, the TP component has the capability to switch engines dynamically (i.e., to change the transport protocol algorithm) in order to adapt to QOS variations.

TP engines ensure efficient delivery of data and perform "in-flow" QOS monitoring; i.e., monitoring on a fast time-scale. Each TP engine also ensures that end-to-end QOS is provided to a next engine bound in series therewith. Every TP engine also preferably implements flow control, encapsulation and de-capsulation, scheduling of channels, segmentation and re-assembly, buffering, multicasting and QOS monitoring, and QOS adaptation mechanisms. A variety of different TP engines can be used with the present invention including, but not limited to, qStack, kStack, TCP, RTP/RTCP, XTP and TP4 engines. Thus, for example, if a particular TP engine (e.g., RTP/RTCP) is not providing a desired QOS, the invention can switch to a different TP engine (e.g., qStack) which better meets the QOS requirements, with or without also switching a current NetP engine.

2.3 Transport Multiplexer

The multiplexer ("MuxP") component provides the capability to multiplex multiple media streams into a single channel. This capability is particularly useful when multiple media processors of a single multimedia application are simultaneously active, as described, for example, in ISO/IEC 14496-6 CD, "Delivery Multimedia Integration Framework, DMIF" (May 1998). This capability is also useful when many

short term channels are established and released on a common host, as is the case during Web browsing. Examples of MuxP engines which may be used with the present invention include, but are not limited to, DMIF FlexMUX and WebMUX engines. For a detailed description of WebMUX, see "http://info.internet.isi.edu/in-drafts/files/draft-gettys-webmux-00.txt" (visited September 16, 1998), the contents of which are hereby incorporated by reference into the subject application (printout submitted with the subject application).

The MuxP engine performs actual software multiplexing and de-multiplexing for channels that are established within a single transport channel. It is expected that all channels of the MuxP engine experience the same QOS, since they share the same transport channel. However, each channel may have different bandwidth requirements. The MuxP front-end therefore ensures that the total capacity of the established channels is within the capacity region of the MuxP engine. See, e.g., Hyman, J.M., Lazar, A.A., and Pacifici, C., "A Separation Principle Between Scheduling And Admission Control For Broadband Switching", IEEE Journal on Selected Areas in Communications (May 1993); and Lazar, A.A., Ngoh, L.H., and Sahai, A., "Multimedia Networking Abstractions with Quality of Service Gurarantees", Proceedings of the SPIE Conference on High-Speed Networking and Multimedia Computing, San Jose, CA, vol. 2417, pgs. 140-154 (February 6-8, 1995). MuxP also provides the ability to add and to destroy channels with a given QOS, and ensures that the overall QOS budget of the channel is within a negotiated QOS. Typically, there is only one MuxP front-end for all MuxP channels multiplexed into the same transport channel. This prevents the proliferation of front-ends and insures scalability. MuxP is most often bound directly to a TP, but can also be bound directly to a NetP when so ordered by PSB 49.

2.4 Transport Services

The foregoing describes three types of consumer/producer engines which may be located in the data path of a protocol stack (i.e., TP, NetP, and MuxP). This section describes three transport services provided

by the present invention. These include the protocol stack builder ("PSB"), the transport monitor, and the QOS mapper.

2.4.1 Protocol Stack Builder

PSB 49 (see Figure 3) can be a centralized server, but in preferred embodiments of the invention, it is highly distributed, meaning that there is one on each end-system. In operation, PSB 49 creates and dynamically reconfigures protocol stacks by invoking control and management interfaces of the NetP, TP and MuxP front-ends. That is, the PSB is responsible for constructing the transport sections of the data path by binding appropriate ones of the MuxP, TP and NetP engines together. PSB 49 does this by determining which engines should be bound together based on one or more factors including, but not limited to, a desired QOS, a predetermined network connection, a user-input command, a command received from a remote network device, or the like. For example, if it is determined that a particular network connection does not require a TP engine, PSB determines that only MuxP and NetP engines should be bound together.

Thereafter, PSB 49 selects front-ends corresponding to those engines, and binds to the selected front-ends. In the example noted above where only MuxP and NetP engines are used, the PSB selects front-ends 46 and 44, but not 45. Similarly, PSB 49 may select only one front-end/engine, in which case the protocol stack would comprise one engine only. Assuming that more than one front-end/engine pair has been selected, PSB 49 instructs the selected front-ends to bind to corresponding engines in the data path (e.g., engines 41 and 39) so as to create a communication path between itself and the corresponding engines. PSB 49 then orders, via the communication path, the corresponding engines to bind together so as to create a protocol stack within the data path. PSB 49 is generally aware of the variety of engines supported on an end-system, and the order in which those engines may be bound to create a useful protocol stack. Accordingly, PSB 49 will detect if an inappropriate engine combination is desired and, if so, alter the engine combination accordingly

and/or issue a message to the user advising the user of such.

In the same way that PSB 49 selects front-end/engine pairs to create a customized protocol stack, PSB 49 may select media processors, i.e., StreamP front-end/engine pairs, in order to create a customized media processor. That is, as was the case above, PSB 49 selects appropriate media processor front-ends based, e.g., on a user's input, remote instruction, or the like, and then binds the engines corresponding to the selected front-ends. By doing this, PSB 49 creates a customized media processor. This, in turn, may be bound to the customized protocol stack created above merely by binding the appropriate media processor/media transporter engines.

2.4.2 Transport Monitor

The transport monitor, which may be implemented in BC 47, is used to log accounting information for management purposes and to monitor QOS on a slow time scale. Its interface is used by the front-ends to register channels and to log QOS information. For example, when the data path includes a TP engine, then the TP engine registers. When a front-end "de-registers" its channel, the accounting information is logged permanently for future usage by the management system (e.g., for billing, network performance, and dimensioning purposes).

Any type of information related to the amount of data transferred, the duration of a communication, etc., can be logged by the transport monitor. In addition, whenever QOS re-negotiation is successfully performed with a TP, that new QOS is logged, as are delivered and requested QOSs. In this regard, the transport monitor receives QOS monitoring information at regular intervals and evaluates the QOS on a slow time-scale compared to the time-scale of the TP. When the transport monitor detects a large QOS variation for a particular channel, the transport monitor may initiate a transport protocol re-negotiation that can result in PSB 49 dynamically changing the TP engine.

2.4.3 QOS Mapper

The QOS mapper performs translation of QOS specifications between the various protocol stack layers (e.g., between the application, transport and network layers). See Huard, J.-F. and Lazar, A. A., "On QOS Mapping in Multimedia Networks", in Proc. of the 21st IEEE Annual International Computer Software and Application Conference (COMPSAC '97), Washington, D.C. (August 13-15, 1997), the contents of which are hereby incorporated by reference into the subject application as if set forth herein in full. Applications and protocol stack builders at channel establishment and QOS re-negotiation times invoke the mapping capabilities of the QOS mapper. Unlike the PSB, which controls a variety of front-ends, the QOS mapper does not interact with the front-ends directly. The QOS mapper may be implemented in BC 47.

3.0 Integrating the Architecture in its Run-Time Environment

As noted above, by distinguishing between consumer/producer engines and front-ends, separation between signaling/control and transport is achieved. Separation of control and transport is a well-established principle that is recognized in most protocol stack designs and implementations. From service creation and development standpoints, this separation facilitates development of multimedia applications, since the signaling system and transport components can be developed independently. This is in contrast to the conventional approach to software development, in which multimedia application development is usually postponed until all the transport components are made available. Moreover, this separation is advantageous during implementation, since it enables different groups to work on development of the signaling system and transport components in parallel without requiring constant coordination. Finally, as a result of this separation, the likelihood of being able to re-use code is increased.

3.1 Protocol Structure

The structure and implementation of the present invention

differs significantly from traditional transport protocol systems. First, the run-time execution environment of the architecture is the user space; although it could be in the kernel space as well. Second, each front-end is implemented as a class that inherits its protocol structure from a base protocol class called the VirtualPort. In this regard, each front-end is a specialization of the VirtualPort that implements its own control functionality and that re-uses the binding functionality implemented in the base protocol class. The use of inheritance dictates the protocol's structure and facilitates dynamic construction of protocol stacks at run-time.

As noted above, the engines (such as engines 39, 40 and 41 in Figure 5) are located in the data path and, as such, are optimized for speed, memory usage, and buffer management. In preferred embodiments of the invention, the engines are implemented in the C programming language and are built as dynamically loadable libraries; although the invention is not limited to this. In contrast, the front-ends (such as front-ends 44, 45 and 46 in Figure 5) are signaling entities that preferably, but not necessarily, use CORBA for communications with the binding controllers and, in preferred embodiments of the invention, are implemented in the C++ programming language. Of course, the invention is not limited to using CORBA and C++. In fact, any remote invocation mechanism can be used in place of CORBA, such as Java RMI and DCOM, and any programming language can be used in place of C++.

The front-ends and engines run in the same address space. The engines are loaded by the front-ends at run-time using the operating system's dynamic library loading mechanisms. The engines' interfaces (i.e., MTI and qECI) are preferably implemented as procedure dispatch tables. Using dispatch tables serves two purposes. First, they are preferred for interfacing to the front-ends, since a single call can be made to discover the entire set of engine entry points. Second, they enable layered services to be formed and operated with increased efficiency. The functionalities of the front-ends' control API and engines qECI are substantially equivalent. Every front-end control operation (e.g., frontEnd->Op1()) has its equivalent, qECI interface (e.g., engine->op1()). It is the

responsibility of the front-end to convert the CORBA (or, alternatively, Java RMI, DCOM, etc.) parameters to the platform-dependent parameters used by the qECI and invoke the qECI.

3.2 Protocol Stack Builder

The protocol stack builder dynamically creates a variety of protocol stacks tailored to the special needs of an application on a per channel basis. It builds a specialized graph for each channel by instantiating a media transporter engine required for protocol processing in the data path. If the media transporter class in not yet loaded, it loads it into the service daemon. The daemon is able to support multiple engine implementations of each type of media transporter, since each engine is loaded dynamically and provides the required common qECI. A binding controller manages the context (i.e., the session state) of each channel. In this regard, the binding controller maintains the state of the communication session and can tear down the channel upon request to the protocol stack builder.

4.0 Application of the Invention: XDMIF

MPEG-4 is an emerging international standard that implements coding of audio and video data using object description techniques. See ISO/IEC 14496-6 CD, "Delivery Multimedia Integration Framework, DMIF", cited above. This approach to coding allows multimedia applications to compose complex scenes dynamically from one or more elementary multimedia streams or synthetic objects. The information used to construct such scenes includes a logical structure of the scene, spatial and temporal information of audio and video objects that comprise the scene, and object attributes and graphics primitives. Such information is typically carried within potentially hundreds of data channels. This calls for the establishment and release of numerous short-term channels with an appropriate QOS at a high rate. Traditional methods of signaling, however, are not adequate to meet these demands because of the high overhead introduced.

Accordingly, MPEG-4 specifies a general application and transport delivery framework called the Delivery Multimedia Integration Framework ("DMIF"). The main purpose of DMIF is to hide the details of the transport network from the user, and to ensure signaling and transport interoperability between end-systems. In order to keep the user unaware of the underlying transport details, MPEG-4 defines an interface between user level applications and the DMIF. This interface is called the DMIF Application Interface ("DAI"). The DAI provides the required functionality for realizing multimedia applications with QOS support. Through the DAI, a user may request service sessions and transport channels without regard to a selected communication protocol. Furthermore, the DAI shields legacy applications from new delivery technologies, since it is the responsibility of the underlying DMIF system to adapt to these new transport mechanisms.

Figure 11 depicts the system architecture of "XDMIF" (specifically XDMIF, version 2), which is an implementation of DMIF, and in particular the DAI, using the present invention. Specifically, XDMIF is implemented within the framework of XBIND, which is a broadband kernel for multimedia networks, and illustrates some of the interactions and components involved during the creation of a multimedia service. See http://comet.ctr.columbia.edu/xbind/overview.html (visited August 26, 1998) for a description of XBIND, the contents of which are hereby incorporated by reference into the subject application (printout submitted with the subject application). In brief, the XBIND broadband kernel is an open programmable networking platform for creating, deploying, and managing sophisticated next-generation multimedia services. It is conceptually based on XRM – a reference model for multimedia networks (See Lazar, A. A., "Programming Telecommunication Networks", cited above, for details). The term 'broadband kernel' is used to draw analogy to the operating system-like functionality that the system must support, namely those of a resource allocator and an extended machine.

As shown in Figure 11, the architecture of XDMIF is

comprised of a transport plane and a control plane, labeled U-plane and C-plane, respectively. XDMIF's C-plane functionality is implemented around the DMIF Network Interface ("DNI"). See ISO/IEC 14496-6 CD, "Delivery Multimedia Integration Framework, DMIF", cited above, for a detailed description of the DNI. The transport architecture of XDMIF allows for the dynamic creation of protocol stacks by binding transport components at run-time. The architecture further permits multiple implementations of a protocol stack layer to co-exist within a single application and for new capabilities to be added without having to modify existing architectural components. In this way, the same application can "interoperate" across ATM, Internet, mobile and telephone networks.

In more detail, the MuxP engines shown in Figure 11 multiplex data with similar QOS requirements into one transport channel, and also perform appropriate demultiplexing. Specifically, they implement the transport capabilities of the conventional DMIF FlexMux, which is the multiplexer for MPEG-4 elementary streams. The TransMux layer of the conventional DMIF is replaced by combinations of TP and TP engine and NetP and NetP engine pairs, as shown in the figure. These can be instantiated, for example, for interworking with ATM Forum or IP based networks.

XBIND provides the QOS framework needed by XDMIF for QOS management. In particular, the protocol stack builder of the present invention must ensure that MuxP has the needed resources available in its transport channel before establishing a new DMIF FlexMux channel. If, in order to satisfy a new channel request, an insufficient amount of resources is available at the FlexMux layer, the XDMIF C-plane requests a new network connection from the network connection controller ("NCC"). Thereafter, the protocol stack builder constructs a new stack in the manner described above using some, or all, of the NetP, TP and MuxP engines.

4.1 Video Conferencing

Figures 12 and 13 show graphical user interfaces ("GUIs") for a video conferencing system which may by implemented by XBIND.

The details pertaining to the operation of the application that controls the video conferencing system are unimportant for the purposes of the present invention. Rather, suffice it to say that the application "sits" atop the XDMIF U- and C-planes, as shown in Figure 11. Audio and video data (e.g., MPEG-4 data) is passed between instances of this application running on various network nodes via these layers and a protocol stack.

As shown in Figure 12, GUI 70 displays video images 71 and 72 of persons at the various remote network nodes based on received data. In the event that one or both of the images degrades in quality or, alternatively, if a change in image quality is desired for some other reason, the present invention can be invoked to reconfigure the device's protocol stack. To this end, the video conferencing application includes an interface to programmable transport architecture 22 (in this case the U- and C-planes of XDMIF), by which the application passes data to, and invokes, the protocol stack builder.

In this regard, the video conferencing GUI includes "CHANGE" button 74. When activated (e.g., by pointing and clicking using mouse 10), dialog box 76, shown in Figure 13, appears on screen. Dialog box 76 includes information relating to the network connection of a device (e.g., a PC) running the video conferencing application. This information includes network QOS requirements, such as "Stream Priority", "Max Delay" (maximum delay), "Avg Delay" (average delay), "Loss Prob" (probable loss), and "Max Gap Loss" (maximum consecutive packet loss). Once values for the QOS requirements are input to, or changed in, dialog box 76, the invention determines whether it is necessary to reconfigure the device's protocol stack in order to meet the new requirements. If the invention determines that protocol stack reconfiguration is necessary in view of the new QOS requirements, the invention then determines which, if any, of the NetP, MuxP and TP engines should be bound together in order to meet the new QOS requirements. This is done based, e.g., on pre-stored information which is available to the invention and which reflects the effects on QOS of binding together different engines. Accordingly, once the invention determines

which engines should be bound together, it performs all necessary binding in the manner described above.

Of course, reconfiguring the protocol stack in this manner will interrupt the data stream therethrough. However, in preferred embodiments of the invention, the reconfiguring takes place quickly enough so that this interruption cannot be detected by the human eye and ear.

The foregoing is but one application of the invention; many others are possible. In this regard, the present invention has been described with respect to a particular illustrative embodiment. It is to be understood that the invention is not limited to the above-described embodiment and modifications thereto, and that various changes and modifications may be made by those of ordinary skill in the art without departing from the spirit and scope of the appended claims.

APPENDIX

A. Pointer to Procedure Table Entry Point

In the following code, the variable SendFunction is a pointer to a function (i.e., a procedure in an engine's procedure table). It is assigned the address of the function SomeSendFunction. The invocation of SendFunction will execute the routine SomeSendFunction.

```
/* API for sending and receiving data */

typedef long (*PFPORTSEND) (short cid, const char *buf, long size, short attr);
typedef long (*PFPORTRECV) (short cid, char *buf, long *size, short attr);
typedef long (*PFPORTSENDIOV) (short cid, const pPORT_IOV_r iov, short, iovlen, short
attr);
typedef long (*PFPORTRECVIOV) (short cid, pPORT_IOV_r iov, short iovlen, short attr);
typedef long (*PFPORTIOCL) (short cid, long control_cod,...);

long SomeSendFunction (short cid, const char *buf, long size, short attr)
{
        ...
}

PFPORTSEND SendFunction = SomeSendFunction;

long bytesSent = SendFunction(cid, databuffer, buffersize, anAttribute);

/* End */
```

B. Procedure table

As defined herein, an engine's procedure table includes pointers to various functions which can be executed on the engine.

```
/* Media Transfer Inteface ("MTI") */

typedef struct_MTI_r
{
        PFPORTSEND          send;
        PFPORTRECV          recv;
        PFPORTSENDIOV       sendiov;
        PFPORTRECVIOV       recviov;
        PFPORTIOCTL         ioctl'
} MTI_r *pMTI_r;

/* End */
```

C. Indirect Pointer

As defined herein, an indirect pointer comprises a pointer to

a pointer to an engine's procedure table.

```
/* Indirect Pointer */
pMTI_r * ppMIT;
/* End */
```

The foregoing variable "ppMTI" is an indirect pointer. An example of its usage is as follows:

```
/* Start */
long bytesSent = (*ppMTI) -> send(cid, dataBuffer, bufferSize, anAttribute);
/* End */
```

WHAT IS CLAIMED IS:

1. A method of building customized network protocol stacks using plural computer-executable engines located in a data path, at least one computer-executable controller, and plural computer-executable front-ends interposed between the engines and the at least one controller, the method comprising the steps of:

binding the at least one controller to selected front-ends;

instructing the selected front-ends to bind to corresponding engines in the data path so as to create a communication path between the at least one controller and each corresponding engine; and

ordering, via each communication path, the corresponding engines to bind together so as to create a protocol stack within the data path.

2. A method according to Claim 1, wherein the at least one controller, the corresponding engines, and the selected front-ends include standardized interfaces which permit binding.

3. A method according to Claim 1, wherein the plural engines include at least plural types of network provisioning engines, plural types of transport protocol engines, and plural types of multiplexing engines; and

wherein the method further comprises the step of selecting the front-ends based on whether a network provisioning engine, a transport protocol engine, and/or a multiplexing engine is to be bound to the controller.

4. A method according to Claim 3, further comprising the step of selecting which, if any, of the plural types of network provisioning engines, the plural types of transport protocol engines, and the plural types of multiplexing engines are to be bound to the selected front-ends based on at least one factor.

5. A method according to Claim 4, wherein the at least one factor comprises either a user-input command, a remote network command, or a desired quality of service ("QOS") for a network connection which passes through the protocol stack.

6. A method according to Claim 1, further comprising the step of dynamically reconfiguring the protocol stack by:

instructing at least one of the selected front-ends to bind to a different engine so as to create a communication path between the at least one controller and the different engine; and

ordering, via each communication path, the different engine and at least one remaining corresponding engine to bind together so as to create a reconfigured protocol stack in the data path.

7. A method according to Claim 6, wherein the step of dynamically reconfiguring the protocol stack is effected in response to either a user-input command or a degradation in quality of service ("QOS") for a network connection which passes through the protocol stack.

8. A method according to Claim 1, further comprising:

binding the at least one controller to one or more media processor front-ends;

instructing the media processor front-ends to bind to corresponding media processor engines so as to create a communication path between the at least one controller and each corresponding engine;

ordering, via each communication path, the corresponding media processor engines to bind together within the data path; and

outputting data from the bound media processor engines to the protocol stack.

9. A method according to Claim 1, wherein the engines, the at least one controller, and the front-ends are implemented in user space.

10. A method according to Claim 1, wherein at least two of the selected front-ends binds to the same engine, the at least two selected front-ends controlling at least two respective channel segments being transmitted through the same engine.

11. Computer-executable process steps stored on a computer-readable medium, the computer executable process steps to build customized network protocol stacks using plural engines located in a data path, at least one controller, and plural front-ends interposed between the engines and the at least one controller, the computer-executable process steps comprising:

code to bind the at least one controller to selected front-ends;

code, in the controller, to instruct the selected front-ends to bind to corresponding engines in the data path so as to create a communication path between the at least one controller and each corresponding engine; and

code, in the controller, to order, via each communication path, the corresponding engines to bind together so as to create a protocol stack within the data path.

12. Computer-executable process steps according to Claim 11, wherein the at least one controller, the corresponding engines, and the selected front-ends include standardized interfaces which permit binding.

13. Computer-executable process steps according to Claim 11, wherein the plural engines include at least plural types of network provisioning engines, plural types of transport protocol engines, and plural types of multiplexing engines; and

wherein the computer-executable process steps further comprise code to select the front-ends based on whether a network provisioning engine, a transport protocol engine, and/or a multiplexing engine is to be bound to the controller.

14. Computer-executable process steps according to Claim 13, further comprising code to select which, if any, of the plural types of network provisioning engines, the plural types of transport protocol engines, and the plural types of network provisioning engines are to be bound to the selected front-ends based on at least one factor.

15. Computer-executable process steps according to Claim 14, wherein the at least one factor comprises either a user-input command, a remote network command, or a desired quality of service ("QOS") for a network connection which passes through the protocol stack.

16. Computer-executable process steps according to Claim 11, further comprising code to dynamically reconfigure the protocol stack by:

instructing at least one of the selected front-ends to bind to a different engine so as to create a communication path between the at least one controller and the different engine; and

ordering, via each communication path, the different engine and at least one remaining corresponding engine to bind together so as to create a reconfigured protocol stack in the data path.

17. Computer-executable process steps according to Claim 16, wherein the code to dynamically reconfigure the protocol stack is executed in response to either a user-input command or a degradation in quality of service ("QOS") for a network connection which passes through the protocol stack.

18. Computer-executable process steps according to Claim 11, further comprising:

code to bind the at least one controller to one or more media processor front-ends;

code to instruct the media processor front-ends to bind to corresponding media processor engines so as to create a communication

path between the at least one controller and each corresponding engine;

code to order, via each communication path, the corresponding media processor engines to bind together within the data path; and

code to output data from the bound media processor engines to the protocol stack.

19. Computer-executable process steps according to Claim 11, wherein the engines, the at least one controller, and the front-ends are implemented in user space.

20. Computer-executable process steps according to Claim 11, wherein at least two of the selected front-ends binds to the same engine, the at least two selected front-ends controlling at least two respective channel segments being transmitted through the same engine.

21. An apparatus which builds customized network protocol stacks using plural computer-executable engines located in a data path, at least one computer-executable controller, and plural computer-executable front-ends interposed between the engines and the at least one controller, the apparatus comprising:

a memory which stores computer-executable process steps to implement the engines, the at least one controller, and the front-ends; and

a processor which executes the process steps stored in the memory so as (i) to bind the at least one controller to selected front-ends, (ii) to instruct the selected front-ends to bind to corresponding engines in the data path so as to create a communication path between the at least one controller and each corresponding engine, and (iii) to order, via each communication path, the corresponding engines to bind together so as to create a protocol stack within the data path.

22. An apparatus according to Claim 21, wherein the at least one controller, the corresponding engines, and the selected front-ends

include standardized interfaces which permit binding.

23. An apparatus according to Claim 21, wherein the plural engines include at least plural types of network provisioning engines, plural types of transport protocol engines, and plural types of multiplexing engines; and

wherein the processor further executes process steps so as to select the front-ends based on whether a network provisioning engine, a transport protocol engine, and/or a multiplexing engine is to be bound to the controller.

24. An apparatus according to Claim 23,wherein the processor further executes process steps so as to select which, if any, of the plural types of network provisioning engines, the plural types of transport protocol engines, and the plural types of multiplexing engines are to be bound to the selected front-ends based on at least one factor.

25. An apparatus according to Claim 24, wherein the at least one factor comprises either a user-input command, a remote network command, or a desired quality of service ("QOS") for a network connection which passes through the protocol stack.

26. An apparatus according to Claim 21, wherein the processor further executes process steps so as to dynamically reconfigure the protocol stack by:

instructing at least one of the selected front-ends to bind to a different engine so as to create a communication path between the at least one controller and the different engine; and

ordering, via each communication path, the different engine and at least one remaining corresponding engine to bind together so as to create a reconfigured protocol stack in the data path.

27. An apparatus according to Claim 26, wherein the

processor executes the process steps so as to dynamically reconfigure the protocol stack in response to either a user-input command or a degradation in quality of service ("QOS") for a network connection which passes through the protocol stack.

28. An apparatus according to Claim 21, wherein the processor further executes process steps so as (i) to bind the at least one controller to one or more media processor front-ends, (ii) to instruct the media processor front-ends to bind to corresponding media processor engines so as to create a communication path between the at least one controller and each corresponding engine, (iii) to order, via each communication path, the corresponding media processor engines to bind together within the data path, and (iv) to output data from the bound media processor engines to the protocol stack.

29. An apparatus according to Claim 21, wherein the engines, the at least one controller, and the front-ends are implemented in user space.

30. An apparatus according to Claim 21, wherein at least two of the selected front-ends binds to the same engine, the at least two selected front-ends controlling at least two respective channel segments being transmitted through the same engine.

31. An apparatus according to Claim 30, wherein the apparatus comprises one of a desk-top personal computer, a laptop, a digital telephone, a digital television, and video-conferencing equipment.

32. A method of binding a media transporter engine in a data path to a media processor engine, the method using at least one computer-executable controller, and at least one computer-executable front-end interposed between the media transporter engine and the at least one controller, the method comprising the steps of:

binding at least one controller to a selected front-end;

instructing the selected front-end to bind to a corresponding media transporter engine in the data path so as to create a communication path between the at least one controller and the corresponding media transporter engine; and

ordering, via the communication path, the corresponding media transporter engine in the data path to bind to the media processor engine.

33. A method according to Claim 32, further comprising the steps of:

instructing the selected front-end to bind to a different media transporter engine so as to create a communication path between the at least one controller and the different media transporter engine; and

ordering, via the communication path, the different media transporter engine to bind to at least one of (i) the media processor engine, and (ii) another media transporter engine in the data path.

34. A method according to Claim 32, further comprising the steps of:

binding at least one controller to a front-end that corresponds to the media processor engine;

instructing the front-end to bind to a different media processor engine so as to create a communication path between the at least one controller and the different media processor engine; and

ordering, via the communication path, the different media processor engine to bind to at least one of (i) another media processor engine, and (ii) another media transporter engine in the data path.

35. Computer-executable process steps stored on a computer-readable medium, the computer executable process steps to bind a media transporter engine in a data path to a media processor engine, the computer-executable process steps forming at least one computer-executable

controller, and at least one computer-executable front-end interposed between the media transporter engine and the at least one controller, the computer-executable process steps comprising:

code to bind at least one controller to a selected front-end;

code to instruct the selected front-end to bind to a corresponding media transporter engine in the data path so as to create a communication path between the at least one controller and the corresponding media transporter engine; and

code to order, via the communication path, the corresponding media transporter engine in the data path to bind to the media processor engine.

36. Computer-executable process steps according to Claim 35, further comprising:

code to instruct the selected front-end to bind to a different media transporter engine so as to create a communication path between the at least one controller and the different media transporter engine; and

code to order, via the communication path, the different media transporter engine to bind to at least one of (i) the media processor engine, and (ii) another media transporter engine in the data path.

37. Computer-executable process steps according to Claim 35, further comprising:

code to bind at least one controller to a front-end that corresponds to the media processor engine;

code to instruct the front-end to bind to a different media processor engine so as to create a communication path between the at least one controller and the different media processor engine; and

code to order, via the communication path, the different media processor engine to bind to at least one of (i) another media processor engine, and (ii) another media transporter engine in the data path.

38. An apparatus for binding a media transporter engine in a

data path to a media processor engine, the apparatus comprising:

at least one memory which stores at least one computer-executable controller, and at least one computer-executable front-end interposed between the media transporter engine and the at least one controller; and

a processor which executes computer-executable process steps from the memory so as (i) to bind at least one controller to a selected front-end, (ii) to instruct the selected front-end to bind to a corresponding media transporter engine in the data path so as to create a communication path between the at least one controller and the corresponding media transporter engine, and (iii) to order, via the communication path, the corresponding media transporter engine in the data path to bind to the media processor engine.

39. An apparatus according to Claim 38, wherein the processor further executes process steps so as (i) to instruct the selected front-end to bind to a different media transporter engine so as to create a communication path between the at least one controller and the different media transporter engine, and (ii) to order, via the communication path, the different media transporter engine to bind to at least one of (a) the media processor engine, and (b) another media transporter engine in the data path.

40. An apparatus according to Claim 38, wherein the processor further executes process steps so as (i) to bind at least one controller to a front-end that corresponds to the media processor engine, (ii) to instruct the front-end to bind to a different media processor engine so as to create a communication path between the at least one controller and the different media processor engine, and (iii) to order, via the communication path, the different media processor engine to bind to at least one of (a) another media processor engine, and (b) another media transporter engine in the data path.

FIG. 1

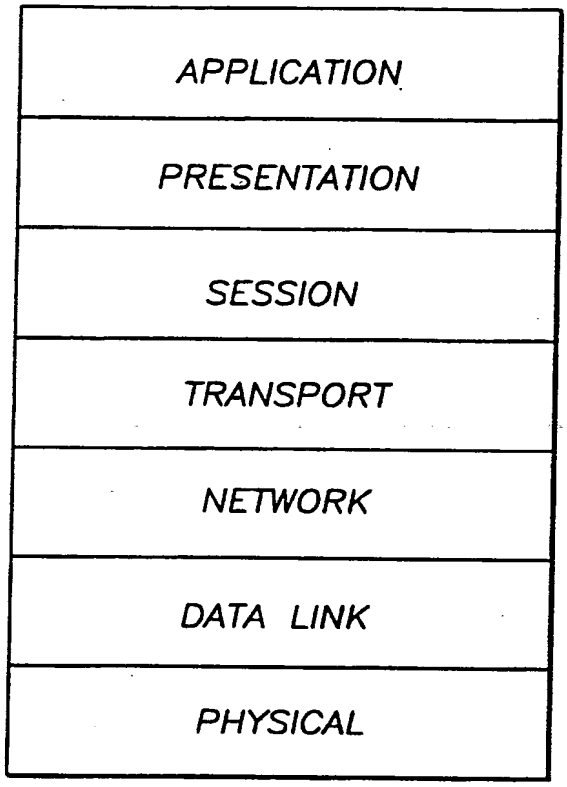


FIG. 2

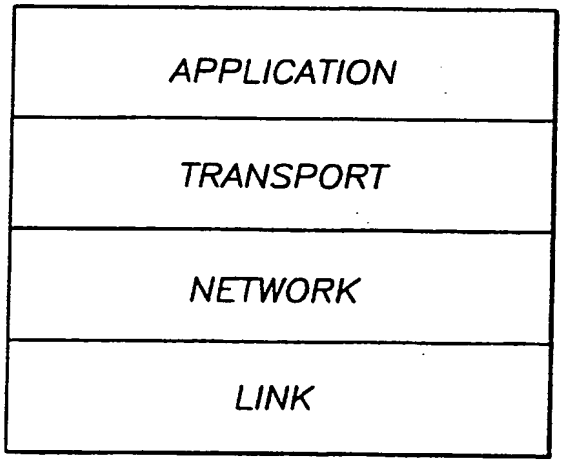
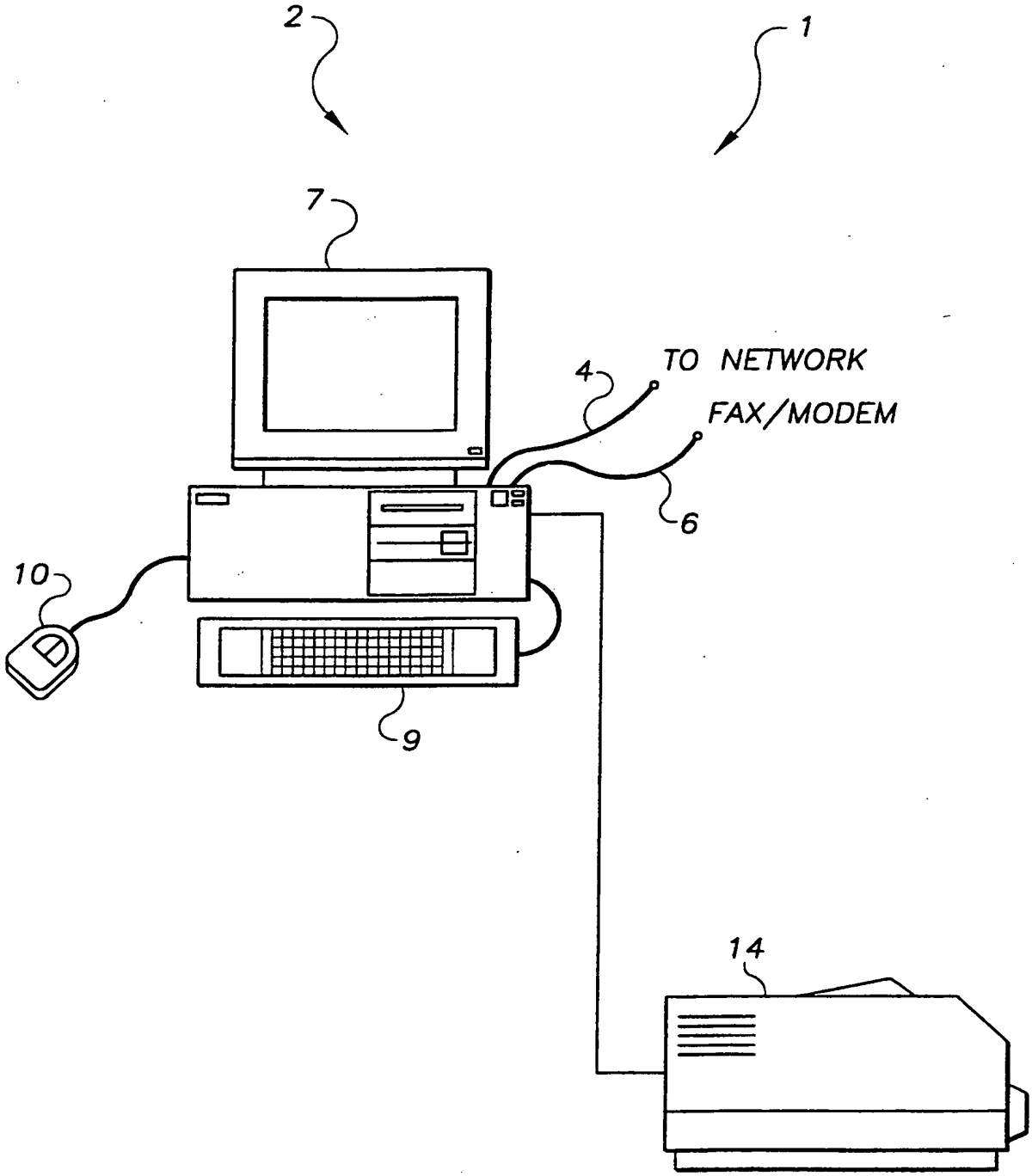


FIG. 3



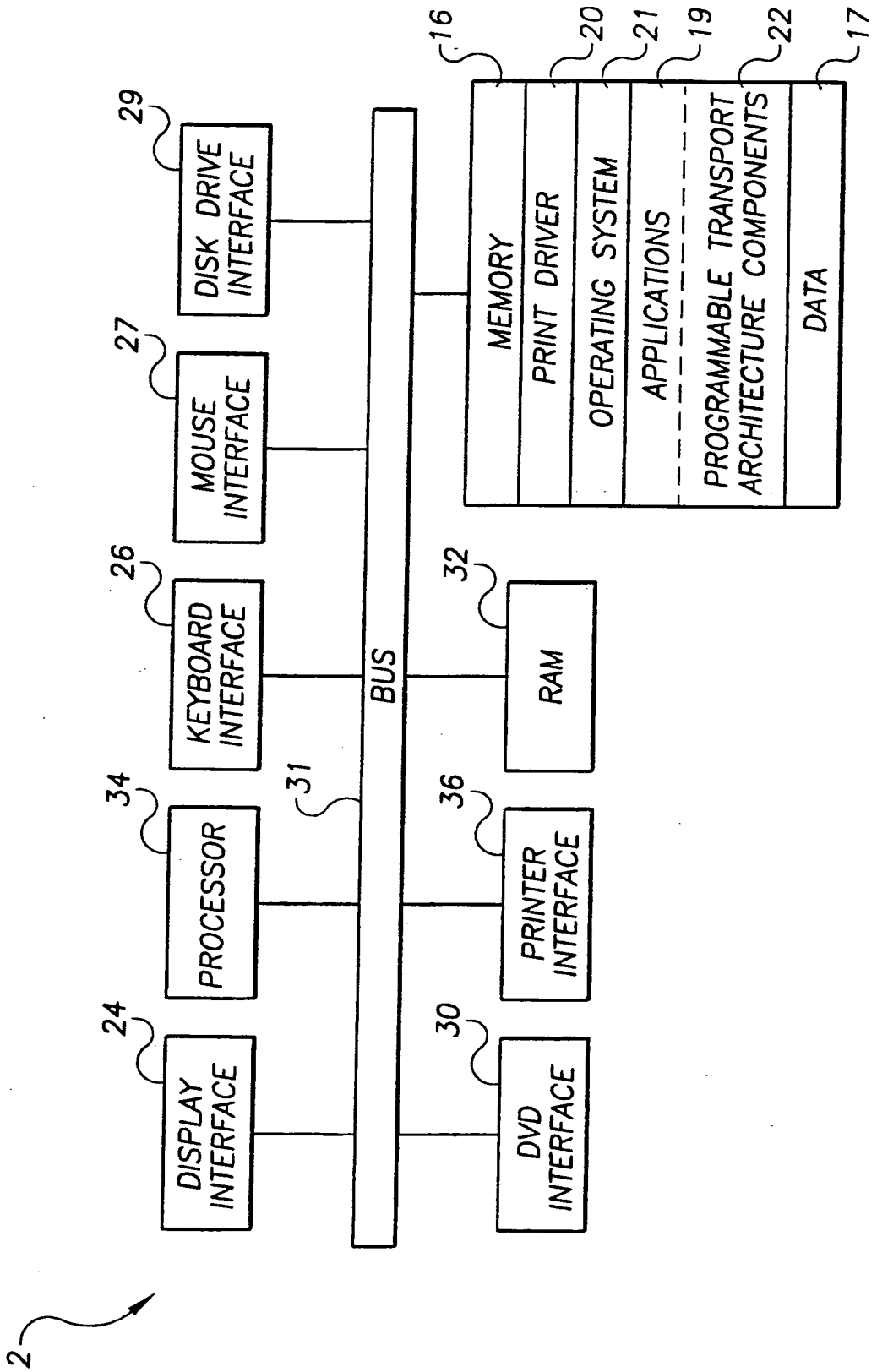


FIG. 4

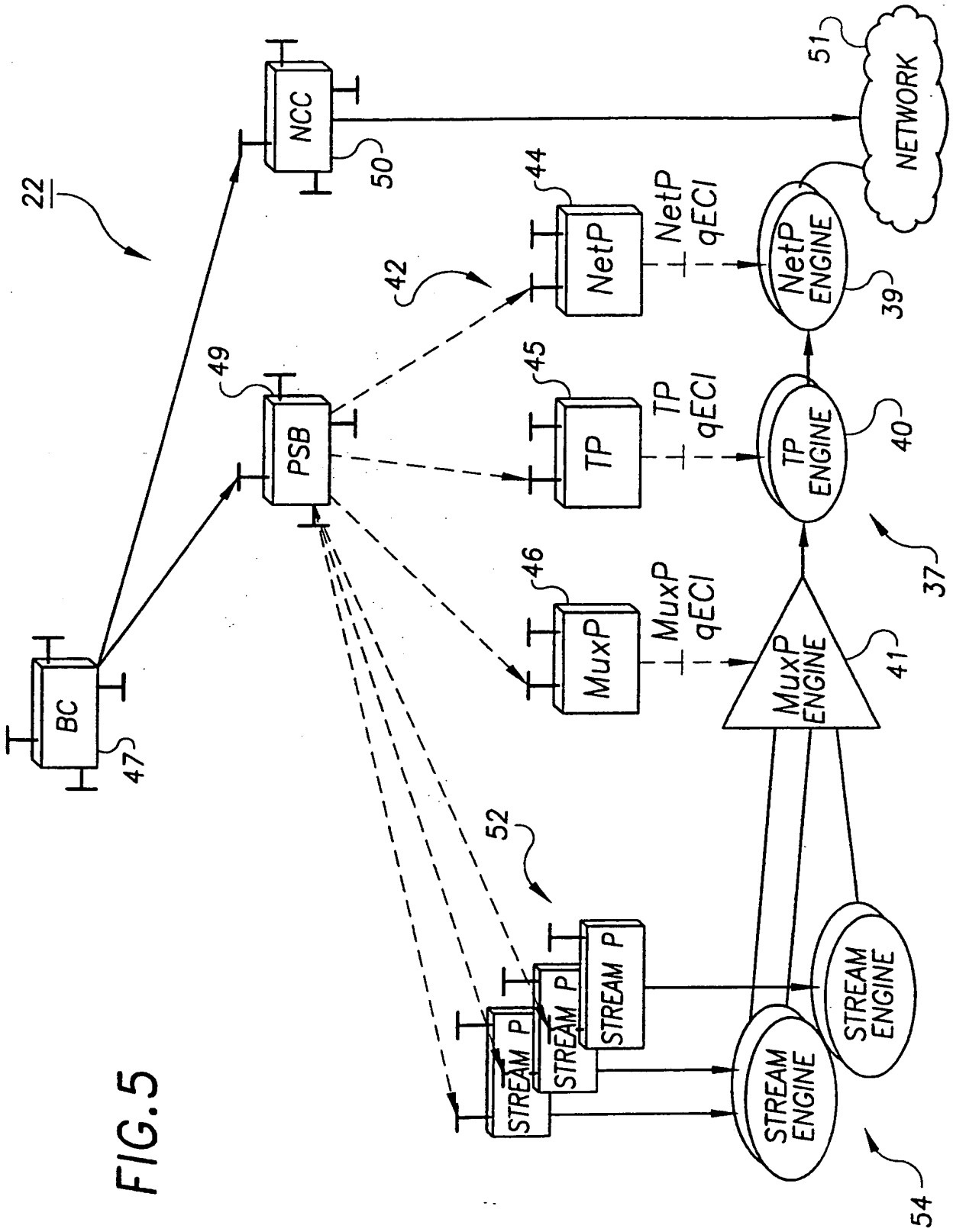


FIG.5

FIG. 6

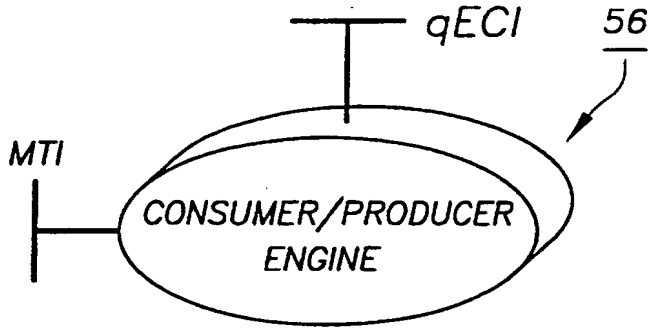


FIG. 8

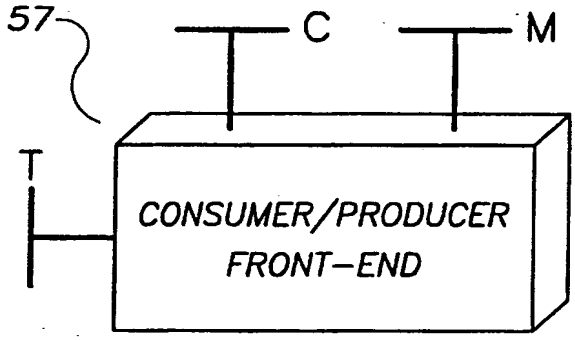
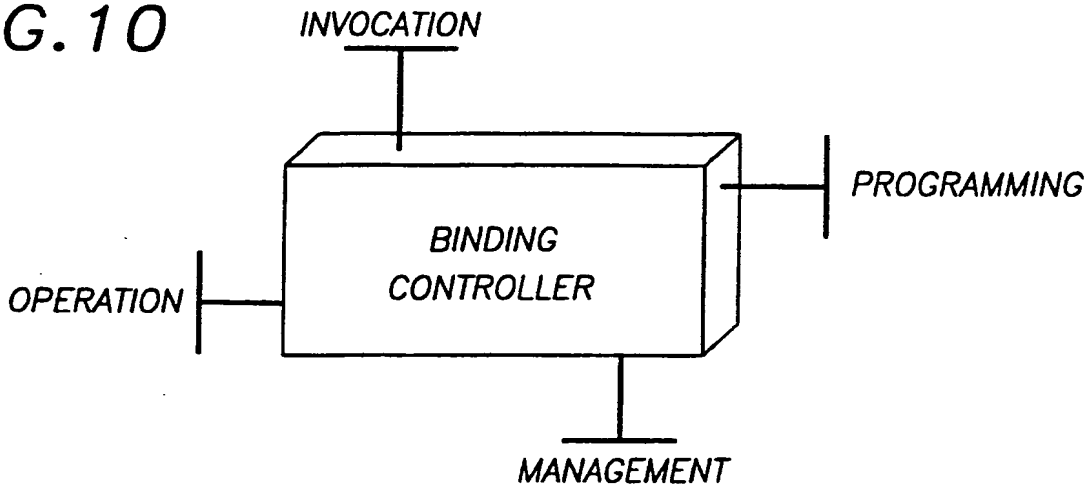


FIG. 10



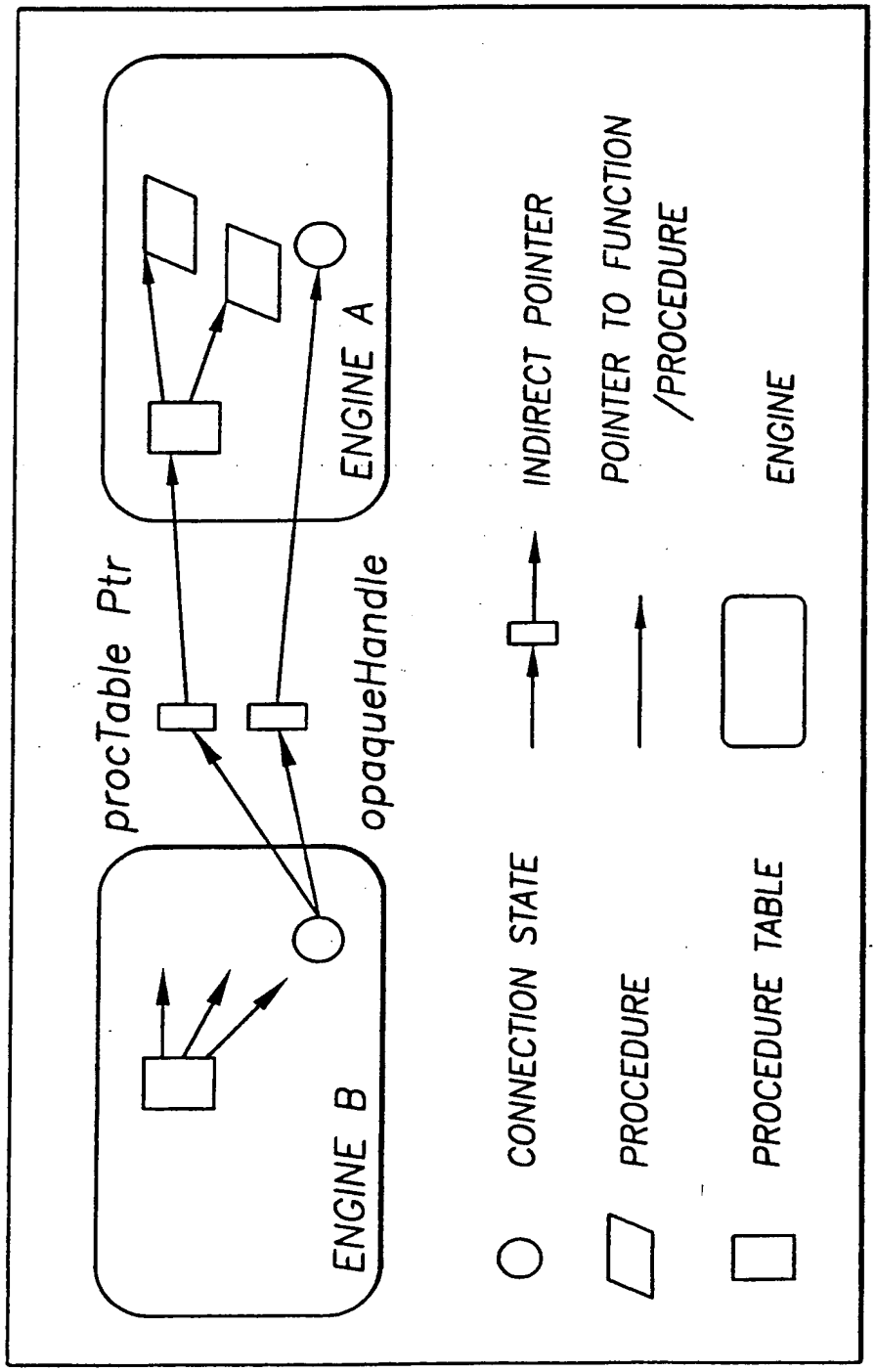


FIG.7

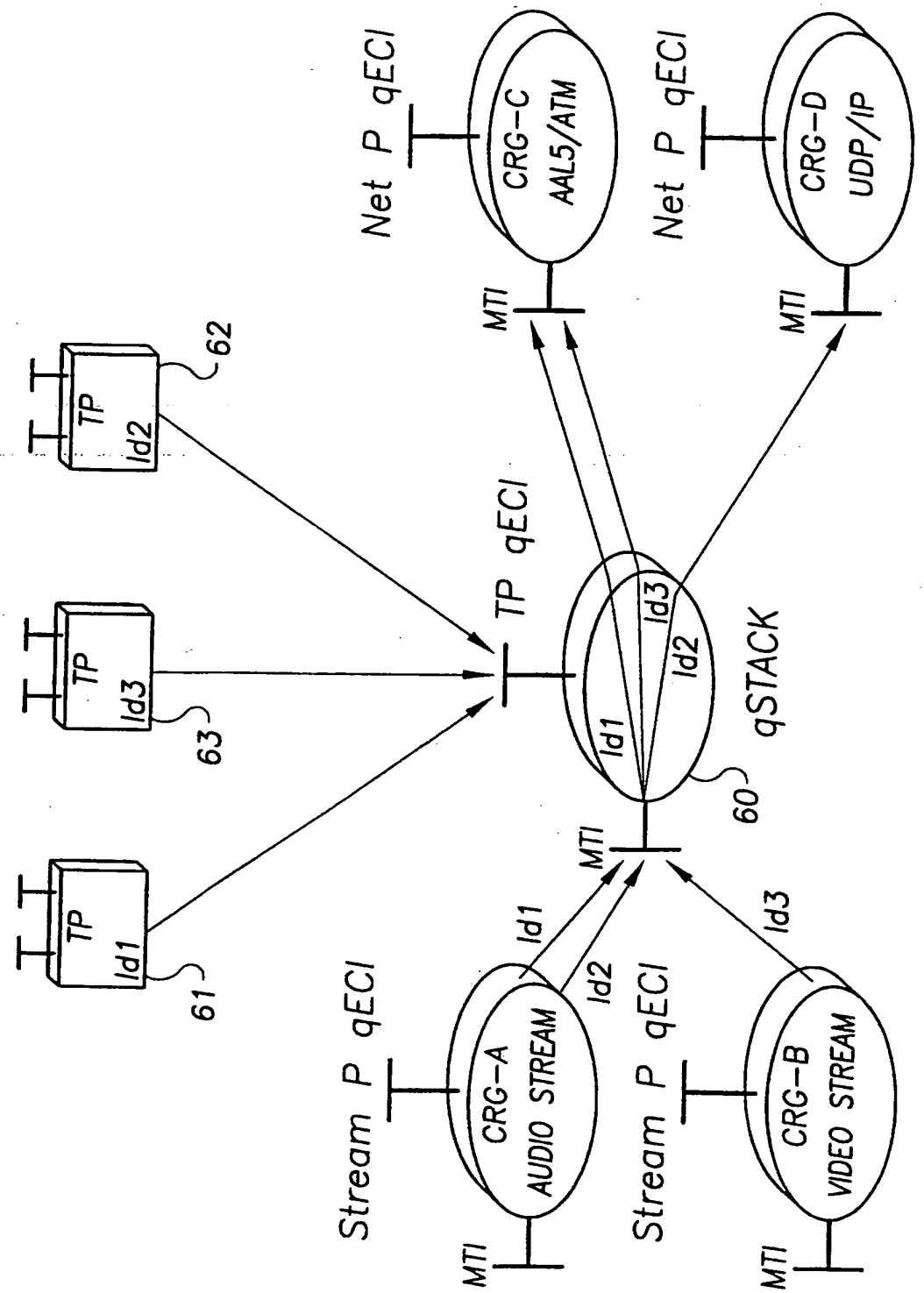


FIG.9

FIG. 11–1

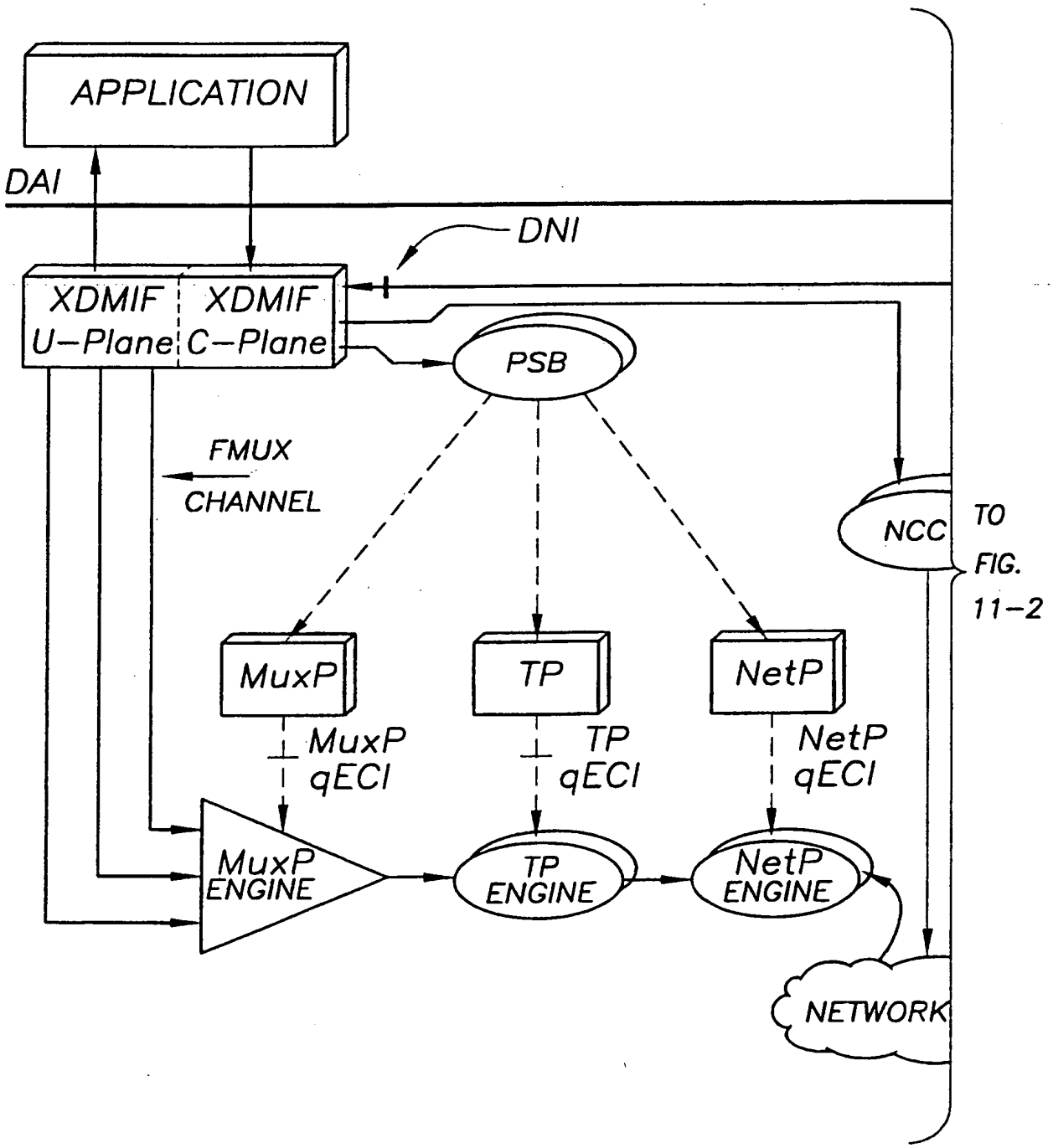
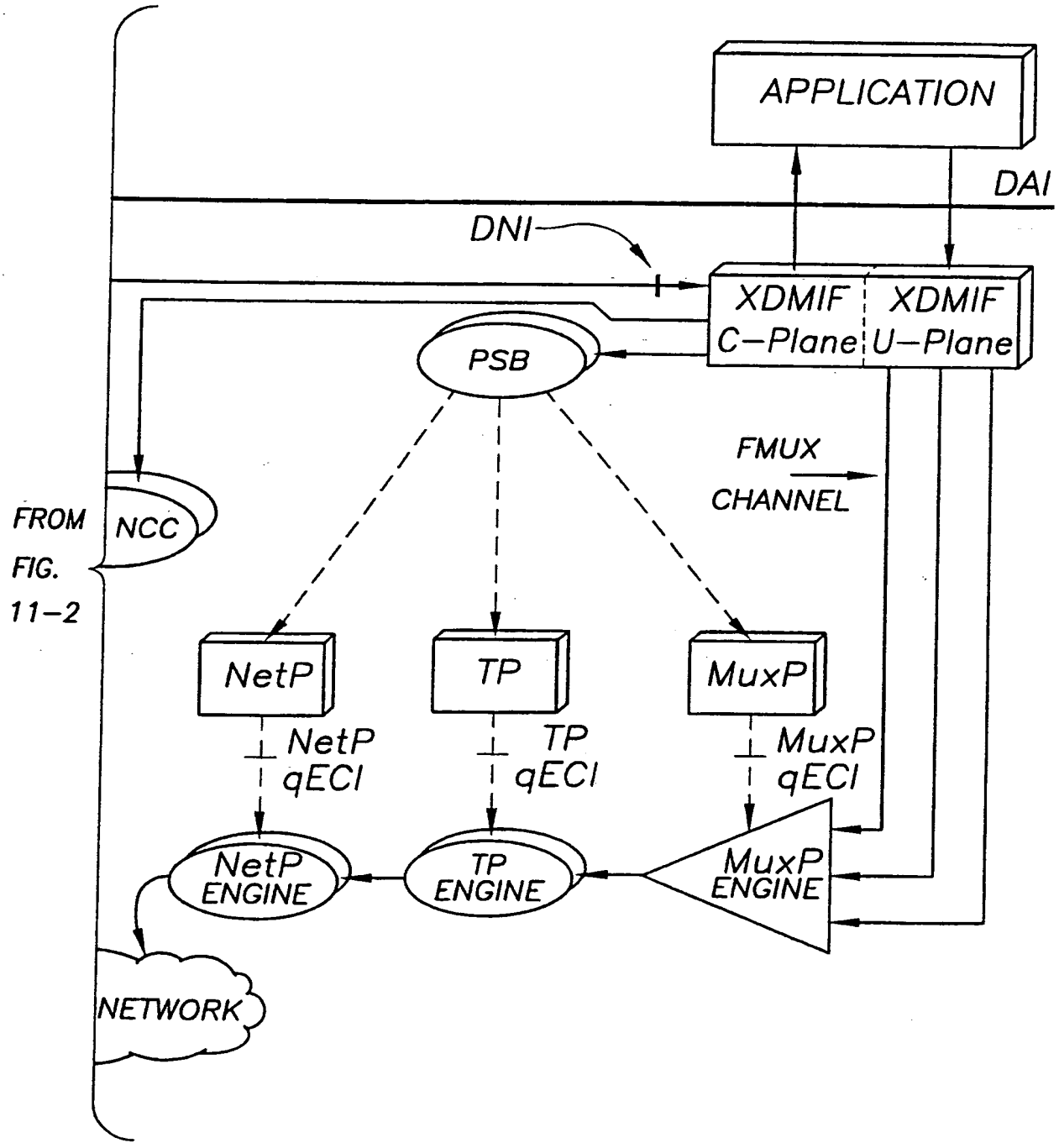


FIG. 11-2



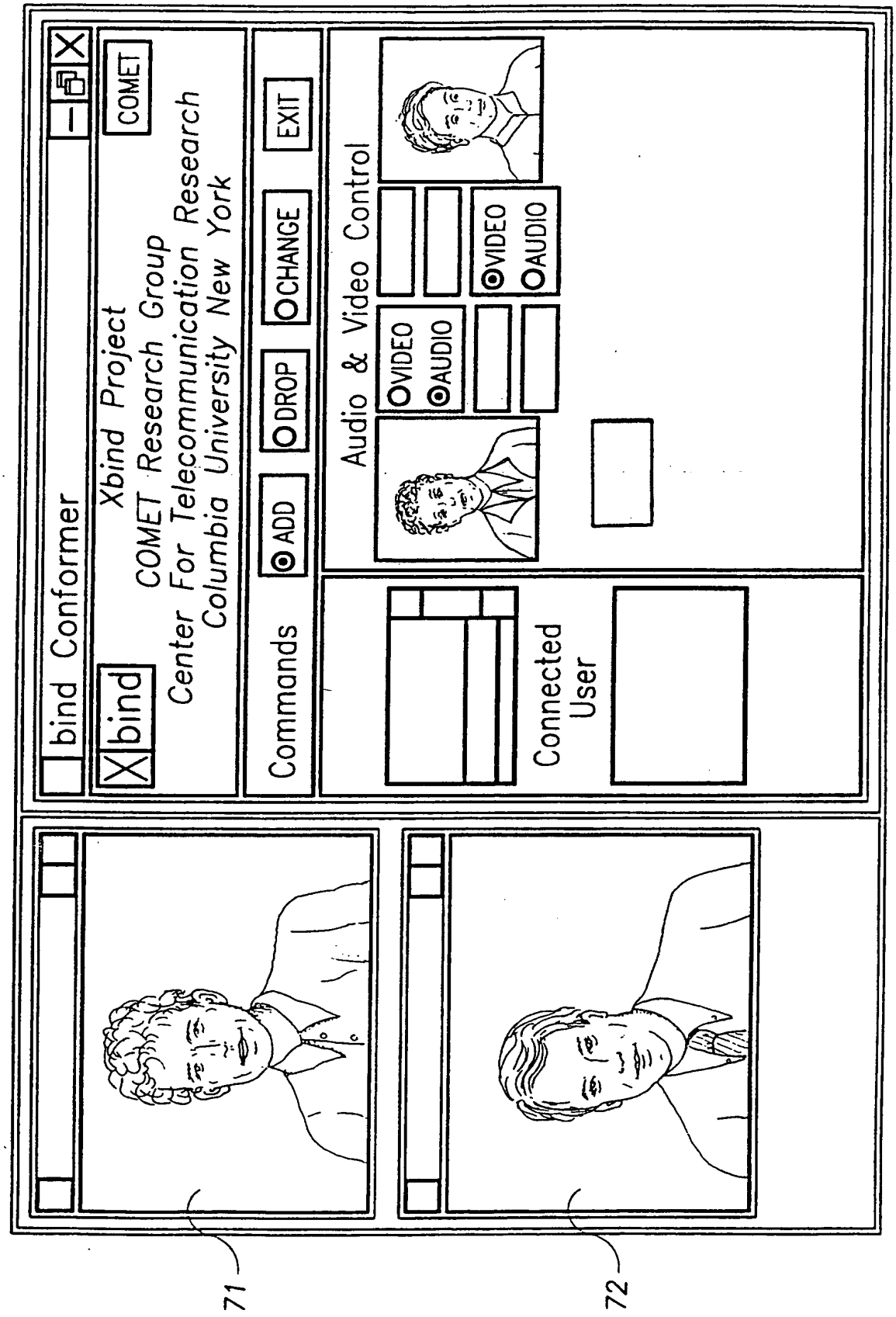


FIG.12

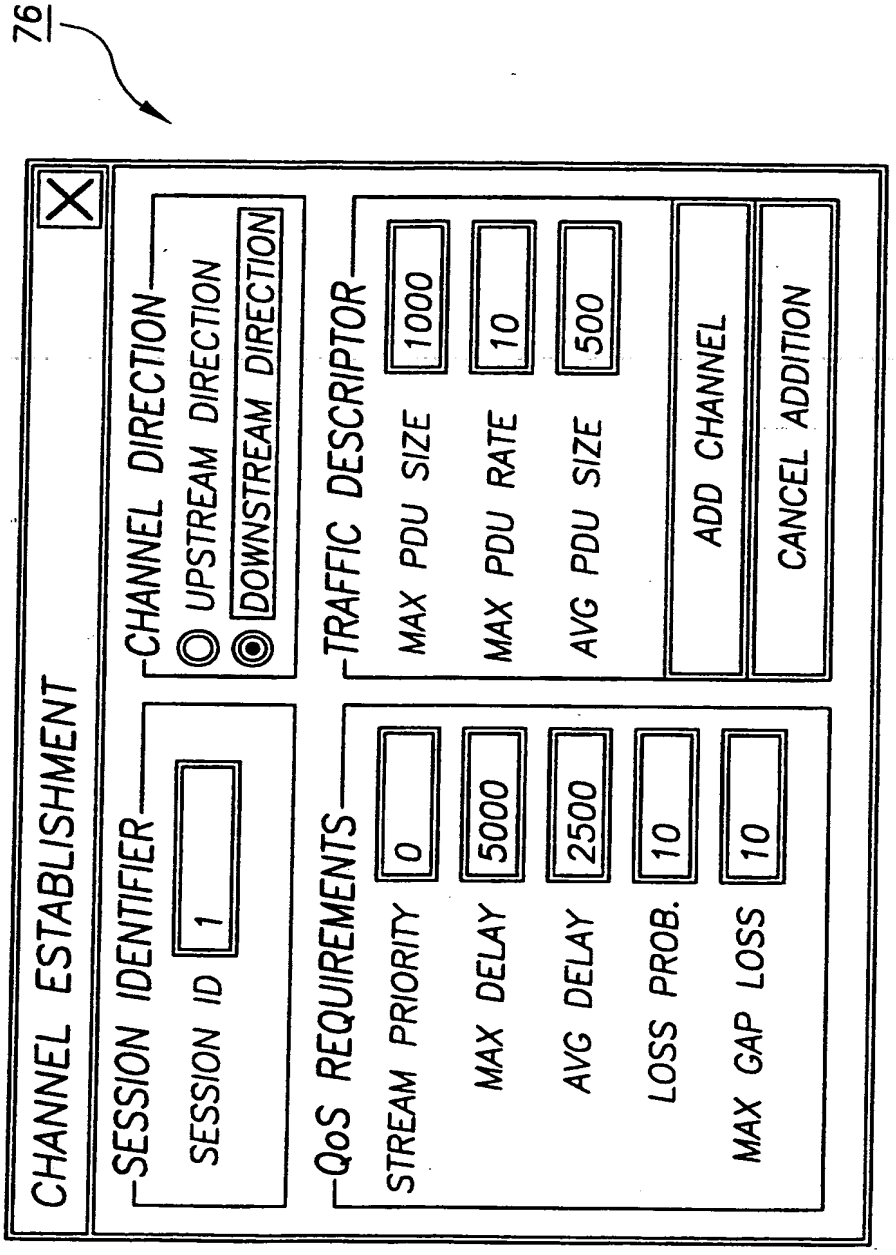


FIG. 13